On the Need for *Practical* Formal Methods[*]

Constance Heitmeyer

Naval Research Laboratory, Code 5546, Washington, DC 20375, USA

Abstract. A controversial issue in the formal methods community is the degree to which mathematical sophistication and theorem proving skills should be needed to apply a formal method. A fundamental assumption of this paper is that formal methods research has produced several classes of analysis that can prove useful in software development. However, to be useful to software practitioners, most of whom lack advanced mathematical training and theorem proving skills, current formal methods need a number of additional attributes, including more user-friendly notations, completely automatic (i.e., pushbutton) analysis, and useful, easy to understand feedback. Moreover, formal methods need to be integrated into a standard development process. I discuss additional research *and* engineering that is needed to make the current set of formal methods more practical. To illustrate the ideas, I present several examples, many taken from the SCR (Software Cost Reduction) requirements method, a formal method that software developers can apply without theorem proving skills, knowledge of temporal and higher order logics, or consultation with formal methods experts.

1 Formal Methods in Practice: Current Status

During the last decade, researchers have proposed numerous formal methods, such as model checkers [6] and mechanical theorem provers [21], for developing computer systems. One area in which formal methods have had a major impact is hardware design. Not only are companies such as Intel beginning to use model checking, along with simulation, as a standard technique for detecting errors in hardware designs, in addition, some companies are developing their own in-house model checkers. Moreover, a number of model checkers customized for hardware design have become available commercially [17].

Although some limited progress has been made in applying formal methods to software, the use of formal methods in practical software development is rare. A significant barrier is the widespread perception among software developers that formal notations and formal analysis techniques are difficult to understand and apply. Moreover, software developers often express serious doubts about the scalability and cost-effectiveness of formal methods.

Another reason why formal methods have had minimal impact is the absence in practical software development of two features common in hardware design environments. First, hardware designers routinely use one of a small group of

[*] This work was supported by the Office of Naval Research and SPAWAR.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1998**	2. REPORT TYPE	3. DATES COVERED **00-00-1998 to 00-00-1998**
4. TITLE AND SUBTITLE **On the Need for Practical Formal Methods**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5546,4555 Overlook Avenue, SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**9**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

languages, e.g., Verilog or VHDL, to specify their designs. In contrast, precise specification and design languages are rarely used in software development.

Second, integrating a formal method, such as a model checker, into a hardware design process is relatively easy because other tools, such as simulators and code synthesis tools, are already a standard part of the design process at many hardware companies. In contrast, in software development, no standard software development environments exist. (Although "object-oriented design" has gained considerable popularity in recent years, this process is still largely informal and what "object-oriented design" means varies considerably from one site to the next.) Moreover, while a number of commercial Computer-Aided Software Engineering (CASE) tools have become available in recent years, few software developers are actually using the tools [18].

This paper distinguishes light-weight formal methods from heavy-duty methods and then describes two broad classes of problems in which formal methods have been useful. It concludes by proposing a number of guidelines for making formal methods more useful in practice.

2 Light-Weight vs. Heavy-Duty Techniques

A formal analysis technique may be classified as either "light-weight" or "heavy-duty". The user of a light-weight technique does not require advanced mathematical training and theorem proving skills. One example of a light-weight technique is an automated consistency checker, which checks a specification for syntax and type errors, missing cases, instances of nondeterminism, and circular definitions (see, e.g., [11, 10]). In contrast, the user of a heavy-duty technique must be mathematically mature, a person skilled in the formulation of formal arguments and clever proof strategies. The most common heavy-weight techniques are mechanical theorem provers, such as PVS [21] and ACL2 [16]. Other techniques which may be classified as heavy-duty are those that automatically generate state invariants from specifications (e.g., see [15, 3]). Such state invariants can be presented to system users for validation or, alternately, can be used as auxiliary invariants in proving new properties from the specifications.

Model checkers fall somewhere between light-weight and heavy-duty techniques. Once the user has defined the state machine model and the property of interest, the model checker determines automatically whether the property is satisfied. One problem with most current model checkers is that the languages available for representing the state machine model and the properties of interest can be difficult to learn. Usually, model checkers require the representation of a property is some form of temporal logic. Expressing certain properties in temporal logic is complex and error-prone, not only for practitioners but for formal methods experts as well. An additional problem with many current model checkers is that sometimes the counterexample produced by model checking is thousands of states long and extremely difficult to understand (see, e.g., [25]).

Fortunately, many of the current problems with model checkers are not inherent in the technology but simply limitations of the current implementations.

In Section 4, I suggest some improvements to model checkers, which should make them easier to use, more automatic, and hence more light-weight. I also suggest how automated, mathematically sound abstraction methods, a topic of current research, can help make model checking more accessible to practitioners.

3 Where Have Formal Methods Been Applied?

We can identify two large classes of problems where formal methods have had utility. First, formal methods have been used to formalize, debug, and prove the correctness of algorithms and protocols. Using formal methods to tackle such problems has proven useful in both hardware and software applications: once its correctness has been shown formally, the algorithm (or protocol) may be used with high confidence in many applications. While model checking has been used to analyze various protocols, such as cache coherence protocols and security protocols (see, e.g., [20]), the deep mathematical reasoning needed to verify most algorithms usually requires a heavy-duty theorem prover.

For example, recently, a microprocessor manufacturer, Advanced Micro Devices (AMD), developed a new algorithm for floating point division on the AMD$_5$86, a new Pentium-class microprocessor. To increase confidence in the algorithm's correctness, AMD hired a team of formal methods experts to verify the AMD$_5$86 microcode that implemented floating point division. The experts used ACL2, an extended version of the Boyer-Moore theorem prover, to formalize the algorithm and to check a relatively deep mathematical proof [16]. After nine weeks, the verification effort was successfully completed. Hiring the formal methods experts to construct the formal proof and mechanically check it using ACL2 was a relatively cheap means for AMD to gain confidence in the AMD$_5$86 microcode for floating point division. Moreover, AMD can confidently reuse the algorithm in future microprocessors.

In a second class of problems, formal methods are used to demonstrate that a specification of a system (or system part) satisfies certain properties, or that a detailed description, such as a design or an implementation, of a system satisfies a given specification. In this latter case, the theorems proven have been referred to as "junk" theorems, since unlike theorems proven about algorithms and protocols, these theorems are not usually of interest outside the system that is being analyzed.

Experience applying formal methods to the requirements specifications of TCAS II [10], a collision avoidance system for commercial aircraft, and WCP [14], a safety-critical component of a U.S. military system, illustrates this second class of problems. In each case, the specification was analyzed for selected properties using both consistency checking and model checking. In both cases, the analysis exposed significant errors [10, 1, 14]. The properties analyzed included both application-independent properties (e.g., a given function is total) and application-dependent properties. In each case, the properties and their proofs were of little interest outside the analyzed systems.

The analysis of such specifications does not normally require deep reasoning. Hence, more light-weight methods, such as consistency checking and model checking, are more cost-effective than heavy-duty theorem proving. Yet, the analyses performed were often quite complex, especially for model checking, due to the enormous size of the state space that needed to be analyzed. Further, the presence in the specifications of logical expressions containing a mixture of boolean, enumerated types, integers, and reals as well as arithmetic and functions made automated consistency checking difficult, especially in the TCAS II specification. Two potential solutions to these problems, both of which are topics of current research, are automated abstraction methods and more powerful decision procedures.

4 Guidelines for Producing a Practical Formal Method

Presented below are a number of guidelines for making formal methods more practical. The objective of these guidelines is to make formal methods light-weight and thus more accessible to software developers. To a large extent, applying these guidelines is simply good engineering practice. However, in other cases, additional research is needed.

4.1 Minimize Effort and Expertise Needed to Apply the Method

To be useful in practice, formal methods must be convenient and easy to use. Most current software developers are reluctant to use current formal methods because they find the learning curve too steep and the effort required to apply a formal method too great. In many cases, deciding not to use a formal method is rational. The time and effort required to learn about and apply a formal method may not be worth the information and insight provided by the method; in some cases, the effort could be better spent applying another method, such as simulation. Suggested below are three ways in which the difficulty of learning and applying a formal method can be reduced.

Offer a language that software developers find easy to use and easy to understand. The specification language must be "natural"; to the extent feasible, a language syntax and semantics familiar to the software practitioner should be supported. The language must also have an explicitly defined formal semantics and it should scale. Specifications in this language should be automatically translated into the language of a model checker or even a mechanical theorem prover.

Our group and others (see, e.g., [14, 22, 7]) have had moderate success with a tabular notation for representing the required system behavior. Underlying this notation is a formal state-machine semantics [11]. Others, such as Heimdahl and Leveson [10], have proposed a hybrid notation, inspired by Statecharts [9], that combines tables and graphics; this notation also has a state-machine semantics.

Specifications based on tables are easy to understand and easy for software practitioners to produce. In addition, tables provide a precise, unambiguous basis for communication among practitioners. They also provide a natural organization which permits independent construction, review, modification, and analysis of smaller parts of a large specification. Finally, tabular notations scale. Evidence of the scalability of tabular specifications has been shown by Lockheed engineers, who used a tabular notation to specify the complete requirements of the C-130J Flight Program, a program containing over 230K lines of Ada code [8].

In addition to tabular notations, other user-friendly notations should also be explored. For example, one group is developing a set of tools which analyze message sequence charts, a notation commonly used in communication protocols [24]. Others are exploring a front-end for model checking based on the graphical notation Statecharts.

Make Formal Analysis as Automatic as Possible. To the extent feasible, analysis should be "pushbutton". One formal technique that is already largely pushbutton is automated consistency checking. To achieve this automation, consistency checking is implemented by efficient decision procedures, such as semantic tableaux [11] and BDDs (Binary Decision Diagrams) [10]. Moreover, a recent paper [23] shows how the Stanford Validity Checker (SVC), which uses a decision procedure for a subset of first-order logic with linear arithmetic, can check the validity of expressions containing linear arithmetic, inequalities, and uninterpreted function symbols.

More progress is needed, however. Specifications of many safety-critical applications contain logical expressions that mix booleans, enumerated types, integers, and reals. How to analyze such mixed expressions efficiently is an unsolved problem. New research is needed that shows how various decision procedures, such as term rewriting, BDDs, and constraint solvers, can be combined to check the validity of such expressions.

Another area where more automation is needed is model checking. Before practical software specifications can be model checked efficiently, the *state explosion* problem must be addressed—i.e., the size of the state space to be analyzed must be reduced. An effective way to reduce state explosion is to apply abstraction. Unfortunately, the most common approach is to develop the abstraction in ad hoc ways—the correspondence between the abstraction and the original specification is based on informal, intuitive arguments. Needed are mathematically sound abstractions that can be constructed automatically. Recent progress in automatically constructing sound abstractions has been reported in [4, 5].

Provide Good Feedback. When formal analysis exposes an error, the user should be provided with easy-to-understand feedback useful in correcting the error. Techniques for achieving this in consistency checking already exist (see, e.g., [12, 23]). As noted above, counterexamples produced by model checkers require improvement. One promising approach, already common in hardware design, uses a simulator to demonstrate and validate the counterexample.

4.2 Provide a Suite of Analysis Tools

Because different tools detect different classes of errors, users should have available an entire suite of tools. This suite may include a consistency checker, a simulator, a model checker, as well as a mechanical theorem prover [13]. The tools should be carefully integrated to work together. This is already happening in hardware design, where simulators, model checkers, equivalence checkers, and code synthesis tools are being used in complementary ways; moreover, some progress has been made in making tools, such as model checkers and simulators, work together [17]. One benefit of having a suite of tools is that properties that have been shown to hold using one tool may simplify the analysis performed with a second tool. For example, validating that each function in a specification is total can simplify subsequent verification using a mechanical prover.

4.3 Integrate the Method into the User's Development Process

To the extent feasible, formal methods should be integrated into the existing user design process. Techniques for exploiting formal methods in object-oriented software design and in software development processes which use semiformal languages, such as Statecharts, should also be explored. How formal methods can be integrated into the user's design process should be described explicitly.

4.4 Provide a Powerful, Customizable Simulation Capability

Many formal methods researchers underestimate the value of simulation in exposing defects in specifications. By symbolically executing the system based on the formal specification, the user can ensure that the behavior specified satisfies his intent. Thus, unlike consistency checking, model checking, and mechanical theorem proving, which formally check the specification for properties of interest, simulation provides a means of validating a specification. In running scenarios through the simulator, the user can also use the simulator to check properties of interest. Another use of simulation is in conjunction with model checking; as suggested above, simulation can be used to demonstrate and validate counterexamples obtained from a model checker.

One important approach to selling formal methods is to build customized simulator front-ends, tailored to particular application domains. For example, we have developed a customized simulator front-end for pilots to use in evaluating an attack aircraft specification. Rather than clicking on monitored variable names, entering values for them, and seeing the results of simulation presented as variable values, a pilot clicks on visual representations of cockpit controls and sees results presented on a simulated cockpit display. This front-end allows the pilot to move out of the world of the specification and into the world of attack aircraft, where he is the expert. Such an interface facilitates evaluation of the specification. Using a commercially available GUI (Graphical User Interface) Builder, one can construct a fancy simulator front-end in a few days.

5 More “Usable” Mechanical Theorem Provers

Although mechanical theorem provers have been used by researchers to verify various algorithms and protocols, they are rarely used in practical software development. For provers to be used more widely, a number of barriers need to be overcome. First, the specification languages provided by the provers must be more natural. Second, the reasoning steps supported by a prover should be closer to the steps produced in a hand proof; current provers support reasoning steps that are at too low and detailed a level. One approach to this problem is to build a prover front-end that is designed to support specification and proofs of a special class of mathematical models. An example of such a front-end is TAME, a “natural” user interface to PVS that is designed to specify and prove properties about automata models [2]. Although using a mechanical provers will still require mathematical maturity and theorem proving skills, making the prover more “natural” and convenient to use should encourage more widespread usage.

6 Conclusions

It is my belief that software practitioners who are not formal methods experts can benefit from formal methods research. However, to do so, they need formal methods that are user-friendly, robust, and powerful. To enjoy the benefits of formal methods, a user does not need to be mathematically sophisticated nor need he be capable of proving deep theorems. One analogy that I heard at CAV ’98 is that beautiful music can come from either a good violin or a highly talented violinist. Light-weight techniques offer software developers good violins. A user need not be a talented violinist to benefit. This is in contrast to heavy-duty techniques where the user needs to be a good violinist.

Formal methods research has already produced a significant body of theory. Moreover, some promising research is currently in progress in automated abstraction and automatic generation of invariants (e.g., [19, 4, 15, 5]). However, to make formal methods practical, good engineering is needed. The user of formal methods should not need to communicate the required system behavior in some arcane language nor to decipher the meaning of obscure feedback from an analysis tool. What a practical formal method should do is liberate the user to tackle the hard problems that humans are good at solving, e.g.,

- What exactly is the required system behavior?
- What are the critical system properties?
- How can I make the specification easy for others to understand?
- What are the likely changes to the specification and how can I organize the specification to facilitate those changes?
- What should the missing behavior be? How can this nondeterminism be eliminated? How do I change the specification to avoid this property violation?

References

1. R. J. Anderson et al. "Model checking large software specifications." *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Eng.*, October 1996.
2. M. Archer, C. Heitmeyer, and S. Sims. "TAME: A PVS Interface to Simplify Proofs for Automata Models." *Proc. User Interfaces for Theorem Provers 1998 (UITP 98)*, Eindhoven, Netherlands, July 13-15, 1998.
3. S. Bensalem, Y. Lakhnech, and Hassen Saïdi. "Powerful techniques for the automatic generation of invariants." *Proc. 8th Intern. Conf. on Computer-Aided Verification, CAV '96*, New Brunswick, NJ, July 1996.
4. S. Bensalem, Y. Lakhnech, and S. Owre. "Computing abstractions of infinite state systems compositionally and automatically." *Proc. 10th Intern. Conf. on Computer-Aided Verification, CAV '98*, Vancouver, BC, Canada, June-July 1998.
5. R. Bharadwaj and C. Heitmeyer. "Model checking complete requirements specifications using abstraction." *Automated Software Eng. Journal* (to appear).
6. E. M. Clarke, E. A. Emerson and A. P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications." *ACM Transactions on Programming Languages and Systems* 8(2):244–263, 1986.
7. S. Easterbrook and J. Callahan. "Formal methods for verification and validation of partial specifications: A case study." *Journal of Systems and Software*, 1997.
8. S. Faulk et al. "Experience applying the CoRE method to the Lockheed C-130J." *Proc. 9th Annual Computer Assurance Conf. (COMPASS '94)*, June 1994.
9. D. Harel. "Statecharts: a visual formalism for complex systems." *Science of Computer Programming* 8(1): 231–274, Jan. 1987.
10. M. P. E. Heimdahl and N. Leveson. "Completeness and consistency analysis in hierarchical state-based requirements." *IEEE Trans. on Software Eng.* SE-22(6), pp. 363-377, Jun. 1996.
11. C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated consistency checking of requirements specifications." *ACM Trans. Software Eng. and Method.* 5(3), 1996.
12. C. Heitmeyer, J. Kirby, and B. Labaw. "Tools for formal specification, verification, and validation of requirements." *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, June 1997.
13. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. "SCR*: A Toolset for specifying and analyzing software requirements." *Proc. 10th Intern. Conf. on Computer-Aided Verification, CAV '98*, Vancouver, BC, Canada, June-July 1998.
14. C. Heitmeyer, J. Kirby, and B. Labaw. "Applying the SCR requirements method to a weapons control panel: An experience report." *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, St. Petersburg, FL, March 1998.
15. R. Jeffords and C. Heitmeyer. "Automatic generation of state invariants from requirements specifications." *Proc. 6th ACM SIGSOFT Symposium Foundations of Software Eng.*, November 1998 (accepted for publication).
16. M. Kaufmann and J S. Moore. ACL2: An industrial strength theorem prover for a logic based on common Lisp. *IEEE Trans. on Software Eng.* SE-23(4), Apr. 1997.
17. R. Kurshan. Formal verification in a commercial setting. *Proc. Design Automation Conference*, June 1997.
18. D. Lending and N. L. Chervany. "CASE tools: Understanding the reasons for non-use," *ACM Computer Personnel* 19(2), Apr. 1998.
19. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. "Property preserving abstractions for the verification of concurrent systems." *Formal Methods in System Design* 6, 1–35, 1995.

20. G. Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR." In *Tools and Algorithms for the Construction of Systems*, Margaria and Steffen, eds., LNCS 1055, Springer-Verlag, 147-166, 1996.
21. Sam Owre, John Rushby, Natarajan Shankar, Friedrich von Henke. "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS." *IEEE Transactions on Software Engineering,* vol. 21, no. 2, pp. 107-125, Feb. 1995.
22. S. Miller. "Specifying the mode logic of a flight guidance system in CoRE and SCR." *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, St. Petersburg, FL, March 1998.
23. D. Y. W. Park et al. "Checking properties of safety-critical specifications using efficient decision procedures." *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, St. Petersburg, FL, March 1998.
24. D. Peled. "A toolset for message sequence charts." *Proc. 10th Intern. Conf. on Computer-Aided Verification, CAV '98*, Vancouver, BC, Canada, June-July 1998.
25. S. T. Probst. "Chemical process safety and operability analysis using symbolic model checking." PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1996.

This article was processed using the LaTeX macro package with LLNCS style